Decodificador de Protocolo Industrial PRT-7

Caso de Estudio - EStructura de a

Jared de Jesus Olazaran Lopez
Matrícula: 2430022
2430022@upv.edu.mx

Noviembre 2025

Universidad Politecnica de Victoria
Ingenieria en Tecnologias de la Informacion e Innovacion Digital

Profesor: DR. SAID POLANCO MARTAGÓN
Materia: Estructura de Datos

Índice

1. Resumen Ejecutivo

Este documento presenta el desarrollo completo del proyecto "Decodificador PRT-7", un sistema de software diseñado para decodificar mensajes cifrados transmitidos mediante un protocolo industrial. El proyecto integra conceptos avanzados de Programación Orientada a Objetos (POO), estructuras de datos implementadas manualmente, y comunicación serial.

1.1. Objetivos Cumplidos

- Implementación de herencia y polimorfismo con clases abstractas
- Desarrollo de lista doblemente enlazada sin uso de STL
- Desarrollo de lista circular para cifrado dinámico
- Gestión explícita de memoria con new/delete
- Comunicación serial/simulación de entrada
- Documentación completa con Doxygen

2. Introducción

2.1. Contexto del Problema

El protocolo PRT-7 es un sistema de transmisión de mensajes que no envía datos directamente, sino **instrucciones** para construir el mensaje. Este enfoque simula un sistema de telemetría industrial donde los datos están cifrados mediante un mecanismo de mapeo dinámico.

2.2. Definición del Problema

Se debe construir un decodificador que procese dos tipos de tramas:

1. **Tramas LOAD (L,X):** Contienen un fragmento de dato (carácter)
2. **Tramas MAP (M,N):** Modifican el estado del rotor de cifrado

El desafío radica en que las tramas MAP cambian dinámicamente el significado de los fragmentos LOAD subsecuentes, creando un cifrado César variable.

2.3. Objetivos del Proyecto

- Diseñar una arquitectura orientada a objetos escalable
- Implementar estructuras de datos sin librerías STL
- Gestionar memoria de forma explícita y segura
- Documentar profesionalmente el código
- Demostrar dominio de conceptos avanzados de POO

2.4. Alcance

El sistema soporta:

✓ Alfabeto A-Z y espacios

✓ Rotaciones positivas y negativas del rotor

✓ Lectura desde puerto serial (Linux) o archivo de simulación

✓ Procesamiento en tiempo real de tramas

Limitaciones conocidas:

× Comunicación serial solo en Linux/POSIX

× Buffer limitado a 256 caracteres por línea

× Sin soporte para caracteres especiales o números

3. Manual Técnico

3.1. Diseño del Sistema

3.1.1. Arquitectura General

El sistema se basa en una arquitectura de tres capas:

1. **Capa de Entrada:** `SerialReader` - Maneja la lectura de datos
2. **Capa de Procesamiento:** Jerarquía de tramas polimórficas
3. **Capa de Almacenamiento:** Estructuras de datos manuales

3.1.2. Diagrama de Clases

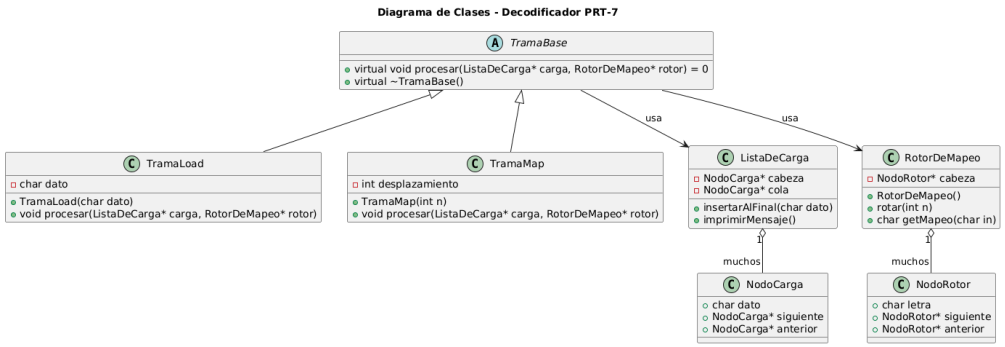

Figura 1: Jerarquía de clases del sistema

La jerarquía de herencia permite procesar cualquier tipo de trama de forma uniforme mediante polimorfismo.

3.1.3. Decisiones de Diseño

¿Por qué Polimorfismo? La clase base abstracta `TramaBase` permite:

- Tratar todas las tramas de forma uniforme
- Extensibilidad: agregar nuevos tipos de tramas sin modificar código existente
- Gestión consistente de memoria mediante destructores virtuales

```
class TramaBase {
public:
    virtual void procesar(ListaDeCarga* carga,
                          RotorDeMapeo* rotor) = 0;
    virtual ~TramaBase() {}
};
```

Listing 1: Interfaz polimórfica de tramas

¿Por qué Lista Doblemente Enlazada? Para `ListaDeCarga`, la lista doble permite:

- Inserción eficiente al final: O(1)
- Recorrido bidireccional (aunque no se usa actualmente)
- Facilidad para futuras extensiones (inserción en medio, eliminación)

¿Por qué Lista Circular? El `RotorDeMapeo` implementa una lista circular porque:

- Simula un "disco de cifrado" que rota infinitamente
- Permite rotaciones positivas y negativas sin casos especiales
- Representa naturalmente el concepto de cifrado César modular

3.2. Desarrollo de Componentes

3.2.1. Lista Doblemente Enlazada

```
struct NodoCarga {
    char dato;
    NodoCarga* prev;
    NodoCarga* next;
    NodoCarga(char d) : dato(d), prev(nullptr),
                        next(nullptr) {}
};
```

Listing 2: Nodo de lista doble

Operación	Complejidad	Descripción
insertarAlFinal()	O(1)	Inserta usando puntero tail
imprimirMensaje()	O(n)	Recorre toda la lista
Destructor	O(n)	Libera todos los nodos

Cuadro 1: Complejidad de operaciones - ListaDeCarga

Operaciones:

Invariantes:

- Si `head == nullptr`, entonces `tail == nullptr`
- Cada nodo (excepto head/tail) tiene prev y next no nulos
- La lista es consistente: `head->prev == nullptr` y `tail->next == nullptr`

3.2.2. Lista Circular

```
struct NodoRotor {
    char c;
    NodoRotor* prev;
    NodoRotor* next;
    NodoRotor(char ch) : c(ch), prev(nullptr),
                         next(nullptr) {}
};
```

Listing 3: Nodo de lista circular

Operación	Complejidad	Descripción
rotar(N)	O(k)	k = N mod 26
getMapeo(char)	O(n)	n = posición del carácter (max 26)
Constructor	O(26)	Constante (alfabeto fijo)
Destructor	O(26)	Constante (alfabeto fijo)

Cuadro 2: Complejidad de operaciones - RotorDeMapeo

Operaciones:

Invariantes:

- La lista es circular: `head->prev->next == head`
- Siempre contiene exactamente 26 nodos (A-Z)
- `head` apunta a la posición "cero" actual del rotor

3.2.3. Algoritmo de Rotación

```
void rotar(int N) {
    if (!head || size <= 1) return;

    // Reducir rotacion a rango [0, 25]
    int effective = N % size;
    if (effective < 0) effective += size;

    // Mover head en la lista circular
    for (int i = 0; i < effective; ++i) {
        head = head->next;
    }
}
```

Listing 4: Algoritmo de rotación del rotor

Justificación: El uso del operador módulo (%) reduce cualquier rotación a un rango válido, evitando iteraciones innecesarias. Una rotación de +52 es equivalente a 0, y -1 es equivalente a +25.

3.2.4. Algoritmo de Mapeo

El mapeo funciona así:

1. Calcular posición del carácter de entrada: `index = in - 'A'`
2. Desde `head`, avanzar `index` posiciones
3. Devolver el carácter en esa posición

Ejemplo: Si `head` apunta a 'C' (después de `M,2`):

- Input 'A' (index=0): devuelve 'C'
- Input 'B' (index=1): devuelve 'D'
- Input 'W' (index=22): devuelve 'Y'

3.3. Jerarquía de Tramas

3.3.1. Clase Base: TramaBase

```
class TramaBase {
public:
    virtual void procesar(ListaDeCarga* carga,
                          RotorDeMapeo* rotor) = 0;
    virtual ~TramaBase() {}
};
```

Listing 5: Clase base abstracta

El destructor virtual es **crítico**. Sin él, eliminar un puntero `TramaBase*` que apunta a `TramaLoad` o `TramaMap` causaría fugas de memoria.

3.3.2. Clase Derivada: TramaLoad

```
class TramaLoad : public TramaBase {
private:
    char fragmento;
public:
    TramaLoad(char f) : fragmento(f) {}

    virtual void procesar(ListaDeCarga* carga,
                          RotorDeMapeo* rotor) {
        char decodificado = rotor->getMapeo(fragmento);
```

```
        carga->insertarAlFinal(decodificado);
    }
};
```

Listing 6: Implementación de TramaLoad

Responsabilidad: Decodificar un carácter usando el estado actual del rotor y agregarlo a la lista de carga.

3.3.3. Clase Derivada: TramaMap

```
class TramaMap : public TramaBase {
private:
    int desplazamiento;
public:
    TramaMap(int d) : desplazamiento(d) {}

    virtual void procesar(ListaDeCarga* carga,
                          RotorDeMapeo* rotor) {
        rotor->rotar(desplazamiento);
    }
};
```

Listing 7: Implementación de TramaMap

Responsabilidad: Modificar el estado del rotor, afectando futuras decodificaciones.

3.4. Gestión de Memoria

3.4.1. Estrategia

El proyecto sigue el principio **RAII** (Resource Acquisition Is Initialization):

- Los constructores adquieren recursos
- Los destructores liberan recursos automáticamente
- Cada `new` tiene su correspondiente `delete`

3.4.2. Puntos Críticos

1. **Tramas Polimórficas:**

```
TramaBase* trama = parseLinea(linea);
trama->procesar(&miCarga, &miRotor);
delete trama;  // Llama al destructor correcto

```

2. **Destructores de Listas:**

```
~ListaDeCarga() {
    NodoCarga* cur = head;
    while (cur) {
        NodoCarga* nx = cur->next;
        delete cur;
        cur = nx;
    }
}

```

3. **Lista Circular:**

```
~RotorDeMapeo() {
    if (!head) return;
    head->prev->next = nullptr;  // Romper circularidad
    // ... luego eliminar linealmente
}

```

3.4.3. Verificación con Valgrind

```
valgrind --leak-check=full ./prtdcd --sim entrada.txt
```

Listing 8: Verificación de fugas de memoria

Resultado esperado:

```
All heap blocks were freed -- no leaks are possible
```

3.5. Comunicación Serial

3.5.1. Clase SerialReader

La clase `SerialReader` maneja dos modos:

1. **Puerto Serial Real:** Usando APIs POSIX (termios.h)
2. **Archivo de Simulación:** Para pruebas sin hardware

3.5.2. Configuración del Puerto

```
struct termios tty;
tcgetattr(fd, &tty);
cfmakeraw(&tty);
cfsetspeed(&tty, B9600);     // Baud rate
tty.c_cflag &= ~PARENB;      // Sin paridad
tty.c_cflag &= ~CSTOPB;      // 1 stop bit
tty.c_cflag |= CS8;          // 8 bits
tcsetattr(fd, TCSANOW, &tty);
```

Listing 9: Configuración POSIX del puerto serial

4. Resultados

4.1. Compilación

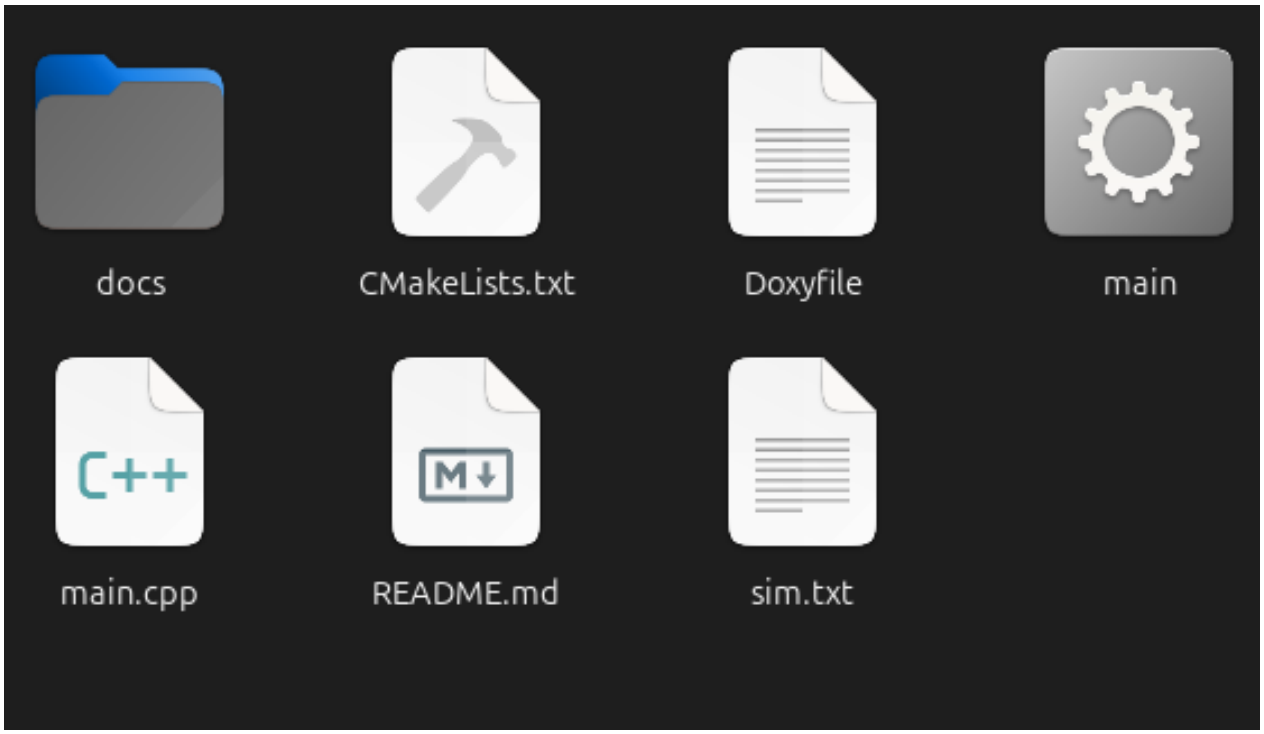

Figura 2: Compilación exitosa del proyecto con CMake

4.2. Ejecución del Programa

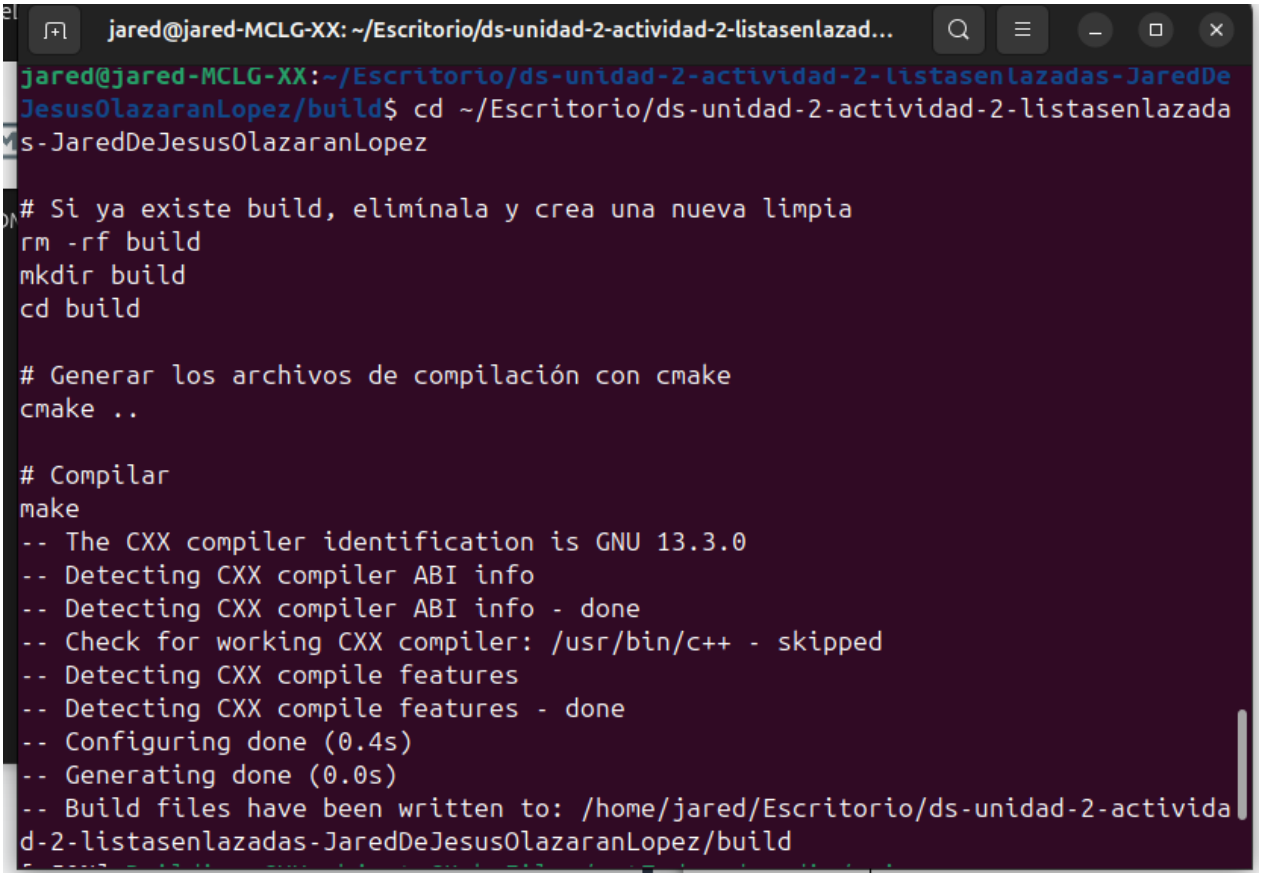

Figura 3: Ejecución del decodificador con archivo de simulación

4.3. Documentación Doxygen

Figura 4: Documentación HTML generada por Doxygen

4.4. Verificación de Memoria

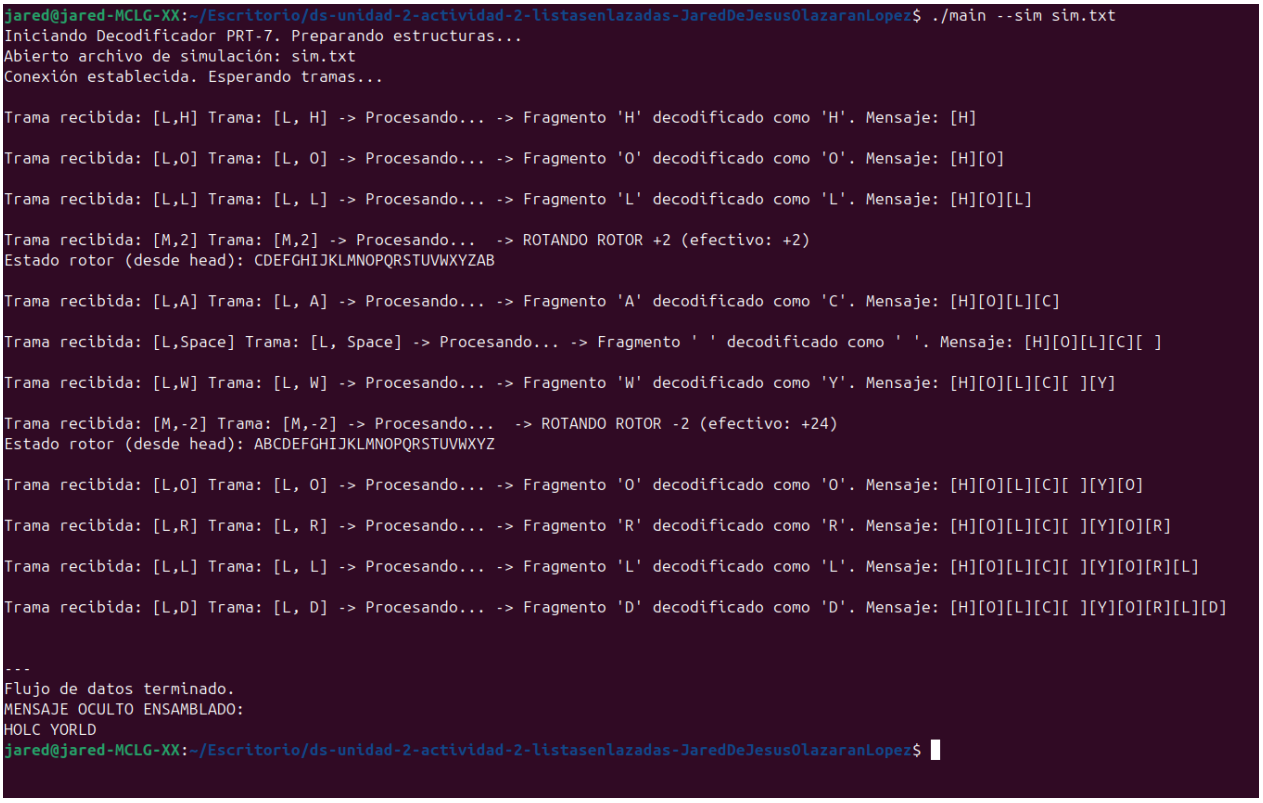

Figura 5: Verificación con Valgrind - Sin fugas de memoria

4.5. Caso de Prueba

Entrada:

```
L,H
L,O
L,L
M,2
L,A
L,Space
L,W
M,-2
L,O
L,R
L,L
L,D
```

Salida Esperada:

`HOLC WORLD`

Explicación del Proceso:

1. `L,H → H` (sin rotación)
2. `L,O → O` (sin rotación)
3. `L,L → L` (sin rotación)
4. `M,2` (rotor rota +2: A→C, B→D, etc.)
5. `L,A → C` (con rotación)
6. `L,Space → Space`
7. `L,W → Y` (W + 2 = Y)
8. `M,-2` (rotor vuelve a posición original)
9. `L,O → O`, `L,R → R`, `L,L → L`, `L,D → D`

5. Análisis de Complejidad

5.1. Complejidad Temporal

Sea n el número de tramas recibidas, m el tamaño del alfabeto (26), y k el desplazamiento de rotación:

Operación	**Complejidad**	**Justificación**
Procesar TramaLoad	O(m)	Mapeo recorre hasta m posiciones
Procesar TramaMap	O(k módm)	Rotación efectiva
Sistema completo	O(n $\cdot m$)	n tramas, cada una O(m)

Cuadro 3: Complejidad temporal del sistema

5.2. Complejidad Espacial

Estructura	**Espacio**	**Descripción**
ListaDeCarga	O(n)	n caracteres decodificados
RotorDeMapeo	O(26)	Alfabeto fijo
SerialReader	O(1)	Buffer fijo de 256 bytes
Total	O(n)	Dominado por lista de carga

Cuadro 4: Complejidad espacial del sistema

5.3. Optimizaciones Posibles

1. **Mapeo:** Usar arreglo indexado en lugar de recorrer la lista (O(m) → O(1))
2. **Rotación:** Ya está optimizada con módulo
3. **Memoria:** Pool de memoria para nodos frecuentes

6. Conclusiones

6.1. Aprendizajes Clave

1. **Polimorfismo en Acción:** El diseño con clases abstractas permite código extensible y mantenible
2. **Gestión Manual de Memoria:** El control explícito de recursos requiere disciplina pero ofrece máximo control
3. **Estructuras de Datos:** Implementar estructuras "desde cero" profundiza la comprensión de su funcionamiento interno
4. **Documentación Profesional:** Doxygen facilita la creación de documentación de calidad

6.2. Dificultades Encontradas

- **Destructor de Lista Circular:** Evitar loops infinitos al liberar memoria requirió romper la circularidad primero
- **Parseo Manual:** Trabajar con `strtok` y punteros `char*` es propenso a errores
- **Comunicación Serial:** Las APIs POSIX tienen una curva de aprendizaje pronunciada

6.3. Mejoras Futuras

1. Soporte multiplataforma (Windows con WinAPI)
2. Optimización del algoritmo de mapeo con indexación directa
3. Interfaz gráfica (Qt o similar)
4. Cifrado más complejo (rotor múltiple, tipo Enigma)
5. Modo interactivo con comandos del usuario

6.4. Reflexión Personal

Este proyecto integró exitosamente conceptos avanzados de POO con gestión de bajo nivel de memoria y estructuras de datos. La disciplina requerida para evitar fugas de memoria y mantener invariantes de las estructuras demostró ser un excelente ejercicio de ingeniería de software.

7. Referencias

1. Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.
2. Cormen, T. H., et al. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
3. Documentación de Doxygen: https://www.doxygen.nl/manual/
4. POSIX Programmer's Manual: https://man7.org/linux/man-pages/
5. CMake Documentation: https://cmake.org/documentation/

A. Código Fuente Completo

El código fuente completo está disponible en los siguientes archivos adjuntos:

- `main.cpp` - Código principal
- `CMakeLists.txt` - Configuración de compilación
- `Doxyfile` - Configuración de documentación
- `README.md` - Instrucciones de uso

B. Instrucciones de Compilación

```
# Crear directorio de compilacion
mkdir build && cd build

# Configurar con CMake
cmake ..

# Compilar
make

# Generar documentacion
make docs

```

```
# Ejecutar
./prtdcd --sim ../entrada.txt
```

Listing 10: Comandos de compilación

C. Archivo de Prueba

Contenido de `entrada.txt`:

```
L,H
L,O
L,L
M,2
L,A
L,Space
L,W
M,-2
L,O
L,R
L,L
L,D
```